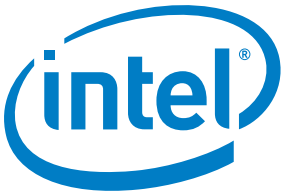

Managed Runtime Speculative Execution Side Channel Mitigations

White Paper

Revision 002
July 2018

Any future revisions to this content can be found at https://software.intel.com/security-software-guidance/ when new information is available. This archival document is no longer being updated.

Document Number: 337313-002

Contents

Revision History

Date	Revision	Description
June 2018	001	Initial release.
July 2018	002	Added RSB mitigation information

§

1.0 Introduction

Side channel analysis methods are techniques that may allow an attacker to obtain secret or privileged information through observing the system that they would not normally be able to access, such as measuring microarchitectural properties about the system. Software components, like managed runtimes that generate code—either via Just-in-Time (JIT) or Ahead-of-Time (AOT) compilation—or code payloads that need to be kept isolated (like applets or programs), should take measures to mitigate these attack methods. This document focuses on mitigating the speculative execution side channel methods *bounds check bypass, branch target injection,* and *speculative store bypass* from the perspective of a managed runtime.

1.1 Document audience

The intended audience for this document is developers of managed runtime environments (for example, JavaScript*, Java*, and C# runtimes). This document provides guidance regarding the JIT and AOT compiler frameworks that exist in these runtimes, and details how the runtime itself should be compiled if mitigation is required. Developers of managed runtime environments should understand the risks of speculative execution side channel attacks (https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf) and the mitigation options (https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf) associated with each variant.

Because this document deals with managed runtime internal requirements, it is not intended to provide guidance for *general developers* who write managed code that runs within a managed runtime, nor is it intended for the users of those managed runtime environments.

1.2 Document scope

The scope of this document includes intraprocess risks and mitigation suggestions for the bounds check bypass, branch target injection, and speculative store bypass side channel analysis methods. The key responsibility to mitigate rogue data cache load falls upon the OS kernel (https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf). The examples and context in this document are primarily focused on user-mode runtimes of languages such as JavaScript, Java, and C#.

The mitigations for bounds check bypass described in this document are focused on direct mitigations for the exploits described by Google* Project Zero

(https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html). This set of mitigations may grow over time as the industry response matures through improved tooling and increased developer awareness.

Intel is also developing further documentation to describe mitigations for the class of vulnerabilities known as side channel analysis methods. Application developers should check https://software.intel.com/en-us/side-channel-security-support for updates on Intel's latest recommendations for mitigating these exploits.

2.0 General mitigations

Side channel analysis methods take advantage of speculative execution that may occur within the same process address space to gain access to secrets. Secrets can be anything that should not be known by other code modules, processes, users, etc. You could separate secrets into different process address spaces and/or use processor-provided security features and technologies for isolation, as described in section 4. In general, we recommend implementing software components that belong to different security domains in different processes. Code that belongs to different security domains should preferably be executed in different process address spaces instead of relying on pointer and code flow control to constrain speculative execution.

2.1 Process isolation

Sandboxed threads in managed runtimes that execute in the same process and rely on language-based security are more susceptible to speculative execution side channel attacks. Process isolation is the preferred mitigation for managed runtimes that can practicably implement it. To implement process isolation, managed runtimes should convert sandboxed threads to sandboxed processes, move secrets into separate processes, and rely on separate address spaces for protection. For example, on a web browser, code from different sites is assumed to belong to different security domains and therefore needs to be in different address spaces (https://www.chromium.org/Home/chromium-security/site-isolation).

However, this strategy could be impractical to implement in some cases. For example, Java Platform*, Enterprise Edition (Java* EE) has a decades-long history of running multiple threads sharing the same address space. For managed runtimes where implementing process isolation is not feasible, you can apply the specific mitigations described in the following sections.

3.0 Managed runtime sandboxes and specific mitigations

A managed runtime is a software environment that simultaneously hosts one or more distinct code payloads (for example, applets, programs, and applications). These applications are sandboxed (logically isolated from each other) via some combination of language constructs and runtime services. The strength of the sandbox isolation varies depending on the particular managed runtime and how that particular managed runtime is used.

For example, assume a Java* runtime supports Java Native Interface* (JNI), which allows arbitrary C code to execute within the managed runtime process. This arbitrary C code can access arbitrary addresses within the managed runtime process, which breaks the sandbox isolation. A Java runtime might prohibit JNI, or might only allow selected, signed code payloads that are trusted to maintain sandbox isolation to execute the C code.

As another example, JavaScript engines should ensure that JavaScript programs from different origins, or security domains, are isolated from one another through strong sandbox isolation.

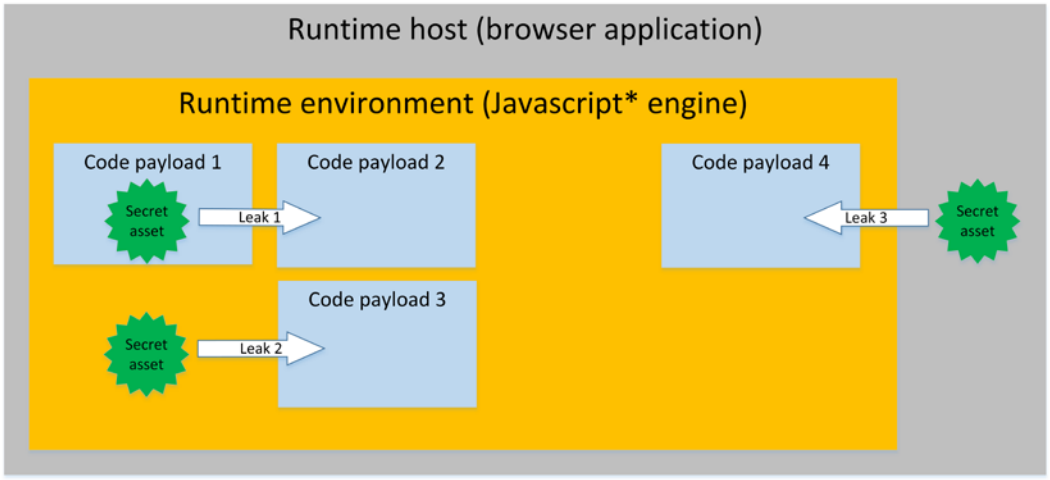

Figure 1: Stereotypical Sandbox Runtime Environment

Figure 1 shows an example of three classes of secret assets (privileged data or code) that belong to different security domains that managed runtimes should protect with sandbox isolation:

- Secret assets in a code payload.
- Secret assets in the runtime environment.
- Secret assets in the runtime host process.

This indicates that several types of attacks must be mitigated to assist with sandbox isolation, as shown with arrows in Figure 1:

1. Code payload 2 detecting secret assets in code payload 1.
2. Code payload 3 detecting secret assets in the runtime environment
3. Code payload 4 detecting secret assets in the runtime host.

If the managed runtime provides sandbox isolation to protect secret assets within the managed runtime, the following general conditions apply:

- Managed runtimes and JIT compilers are generally designed to not trust the code they execute. For example, a browser executing JavaScript from an untrustworthy origin, or security domain.
- The runtime itself generally executes within a host process (for example, a browser) and serves one or more code payloads.
- Code payloads do not mutually trust each other.

If a managed runtime provides sandbox isolation, it should implement side-channel mitigation techniques. Because mitigation involves some overhead, if a given managed runtime only occasionally runs in sandbox isolation mode, managed runtime vendors might consider developing both sandbox-capable and non-sandbox-capable release versions of their runtime, and/or adding options that allow users to choose whether to enable sandboxing or not. On startup, managed runtimes might automatically determine whether sandbox isolation is needed and run the appropriate version of the runtime.

A single application environment, like Node.js*, that only runs trusted sources is an example of a runtime environment that you might wish to run without side-channel mitigations. In this case, a sandbox might not be required within the single-process address space.

Mitigations need to be applied to code executing in the same address space as the secret data: the runtime execution engine (for example, a fast/optimizing JIT or AOT compiler, etc.), the runtime environment itself, and the process hosting the runtime. Note that you may also need to apply the mitigation steps outlined in this document to any libraries used by the runtime and hosting process.

The runtime execution engine mitigations help protect code payloads from side channel attacks by other code payloads. We outline some guidelines for how to insert these mitigations into the runtime execution engine in Section 3.2 for bounds check bypass and Section 3.3 for branch target injection, and 3.4 for speculative store bypass.

The runtime environment itself should include mitigations whenever sandbox isolation is required to help protect the internal runtime state from side channel attacks by code payloads. In general, runtime environments themselves are native applications. Therefore, you should apply the mitigations identified in Sections 3.2, 3.3, and 3.4 of this document to the runtime environment as well.

Additionally, if any inline assembly is included in the managed runtime (for example, for critical lock, allocation, hash, interpreter, etc.), you should analyze the inline assembly as described in Section 3.2, Section 3.3, and Section 3.4 to determine if you need to apply mitigations. Some managed runtimes encode these critical sequences as compiler intermediate representation (IR) and then use the regular JIT/AOT compiler to emit instruction set architecture (ISA) assembly. In this case, managed runtime developers should use a high enough IR level to encode these sequences so that the IR exercises the mitigation insertion path as it is translated into ISA assembly. For example, if the mitigations are inserted during a mid-level IR to low level IR translation and custom snippets are pre-encoded in low level IR, those snippets would not have any mitigations inserted.

To complete sandbox isolation, the process that hosts the managed runtime (for example, the browser that hosts the web runtime) should include mitigations to help protect the host process from being attacked by code payloads. In general, these host applications themselves are native applications. Therefore, you should apply the mitigations outlined in Sections 3.2, 3.3, and 3.4 to the host application as well.

3.1 Short term mitigations

Developers can reduce the precision of timers available to users of the runtime (https://www.mozilla.org/en-US/security/advisories/mfsa2018-01) as a short term mitigation while long term mitigations are being developed. Another short term mitigation that is worth noting is disabling JITs in cases where doing so is acceptable. Disabling JITs reduces the freedom that attackers have to generate vulnerable code sequences, but it is obviously not practical for some environments.

3.2 Bounds check bypass mitigation

Bounds check bypass (https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html) takes advantage of the speculative execution used in processors to achieve high performance. To avoid the processor having to wait for data to arrive from memory, or for previous operations to finish, the processor may speculate as to what will be executed. If it is incorrect, the processor will discard the wrong values and then go back and redo the computation with the correct values. At the program level this speculation is invisible, but because instructions were speculatively executed they might leave hints that a local malicious actor can measure, such as which memory locations have been brought into cache.

Using the bounds check bypass method, malicious actors can use code gadgets ("confused deputy" code) to infer data values that have been used in speculative operations. This presents a method to access data in the system cache and/or memory that the malicious actor should not otherwise be able to read.

3.2.1 Bounds check bypass mitigation overview

In managed runtime languages (for example, Java, JavaScript, and C#), the runtime automatically performs data validation, such as array bounds checks, null checks or type checks. While Spectre variant 1 is known as the *bounds check bypass* vulnerability, code sequences can potentially become data exfiltration gadgets even if there is not an explicit array access or array bounds check. All data validation, including pointer null checks and dynamic object type checks, using conditional branches may need mitigation bounds check bypass. You should watch for further research about recognizing and disrupting gadgets based on this variant. We describe several currently known mitigation techniques in the following sections. When the managed runtime is used in sandbox mode, the bounds check bypass mitigations need to be handled in the interpreter, JIT compiler, and AOT compilers in the language runtime. Developers should do any optimizations in the runtime compilers carefully to avoid inadvertently removing the associated mitigations.

3.2.2 Mitigation through ordering instructions

You can mitigate bounds check bypass attacks by modifying software to constrain speculation in confused deputy code. Specifically, software can insert a barrier that stops speculation between a bounds check and a later operation that could be exploited. The LFENCE instruction can serve as such a barrier. An LFENCE instruction or a serializing instruction will ensure that no later instructions execute, even speculatively, until all prior instructions complete locally. Developers might prefer LFENCE over a serializing instruction because LFENCE may have lower latency. Inserting an LFENCE instruction after a bounds check prevents later operations from executing before the bound check completes. Developers should be judicious when inserting LFENCE instructions. Overapplication of LFENCE can compromise performance.

```
// r10 has the base, r8d has index, r11d is the limit loaded
from memory

// Bounds check

mov   r11d, dword ptr [rsi+0xc]

cmp   r8d, r11d

jae   array_out_of_bounds_error

// Mitigation with speculation stopping barrier

lfence

// Array access

movsx r13d, byte [r10 + r8 + 0x10]
```

Alternatively you could use an LFENCE instruction after the validated load but before the loaded value can be used in a way that creates a side channel (for example, to compute a code or data address), thereby preventing attackers from exfiltrating stolen data.

3.2.3 Runtime/Host mitigation through ordering instructions

Managed runtimes and host application are usually implemented in C/C++. When the managed runtime is used in sandbox environment, to mitigate bounds check bypass through ordering instructions, you should also consider applying the mitigations to the runtime itself, the hosting process, and the libraries used by the runtime.

The `_mm_lfence()` compiler intrinsic can be used for this purpose. It issues an LFENCE instruction and also tells the compiler that memory references may not be moved across that boundary. For example:

```
#include <emmintrin.h>

                ...

 if (user_value >= LIMIT)

       return ERROR;

 _mm_lfence();     /* manually inserted by developer */

 x = table[user_value];

 node = entry[x]
```

In this example, the LFENCE helps ensure that the loads do not occur until the bounds check condition has actually been completed. The memory barrier prevents the compiler from reordering references around the LFENCE, which would break the protection. GCC, Intel® C/C++ Compiler, LLVM, and the Microsoft* Visual* C++ (MSVC) compiler all support generating LFENCE instructions for 32- and 64-bit targets when you manually use the `_mm_lfence()` intrinsic in the required places.

A comprehensive mitigation approach to bounds check bypass (https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html) is proposed on the LLVM mailing list describing compiler assisted mitigation technique for LLVM.

Runtimes/hosts compiled using MSVC compiler could also use compiler assisted mitigation for bounds check bypass. We provide a few observations below:

- All versions of MSVC v15.5 and all previews of MSVC v15.6[1] provide the `/Qspectre` switch, which automatically inserts LFENCE barriers when the

[1] Refer to Microsoft's public documentation for other options:
- https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/

compiler detects code that is vulnerable to bounds check bypass. This switch is effective only on optimized code (for example, `/O1` or `/O2`, but not `/Od`). MSVC v15.7 compiler from preview 3 onwards support the `/Qspectre` switch in `/Od` mode as well.

- For performance-critical blocks of code where you know that mitigation is not needed, you can use `(__declspec( spectre(nomitigation) )` to selectively disable the mitigation while compiling with the `/Qspectre` flag.
- It is important to note that the automatic analysis performed by MSVC and other compilers does not guarantee that the compiler will detect and mitigate all possible instances of bounds check bypass by inserting LFENCE barriers[2]. To ensure that bounds check bypass is fully mitigated, manually insert LFENCE barriers by using `_mm_lfence()` intrinsic in the appropriate places.

3.2.4 Mitigation through index or data clipping

Other instructions (such as CMOVcc, AND, ADC, SBB and SETcc) can also be used to mitigate bounds check bypass attacks by constraining speculative execution on current family 6 processors (Intel® Core™, Intel Atom®, Intel® Xeon® and Intel® Xeon Phi™ processors). Intel will release further guidance on the usage of instructions to constrain speculation if future processors with different behavior are released.

Example for JIT/AOT with CMOVcc to sanitize index:

```
// r10 has the base, r8d has user index, the limit is in
memory at rsi+0xC, r9d will have sanitized index before array
access

// set final index to 0

xor   r9, r9

// Bounds check

cmp   r8d, dword ptr [rsi+0xc]

jae   array_out_of_bounds_error

// Mitigation with Cmovcc: use input index if bounds check
succeeds, otherwise use 0

cmovb r9d, r8d

// Array access
```

- https://docs.microsoft.com/en-us/cpp/build/reference/qspectre
- https://docs.microsoft.com/en-us/cpp/cpp/spectre

[2] Spectre Mitigations in Microsoft's C/C++ Compiler

- https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

```
movsx r13d, byte [r10 + r9 + 0x10]   ; r9 would have been
sanitized to zero for out of bounds index
```

Example for JIT/AOT with CMOVcc to sanitize data:

```
// r10 has the base, r8d has user index, the limit is in
memory at rsi+0xC, r9d will have sanitized data

// set final data to 0

xor   r9, r9

// Bounds check

cmp   r8d, dword ptr [rsi+0xc]

jae   array_out_of_bounds_error

// Array access and get data into r13d

movsx r13d, byte [r10 + r8 + 0x10]

// Mitigation with Cmovcc: use data read from array if bounds
check succeeds, otherwise use 0

cmovb r9d, r13d   ; data in r9d would have been sanitized to
zero for out of bounds index
```

Example for JIT/AOT with SBB:

```
// r10 has the base, r8d has user index, the limit is in
memory at rsi+0xC, r9 will have sanitized index before array
access

// Bounds check

cmp   r8d, dword ptr [rsi+0xc]

jae   array_out_of_bounds_error

// Mitigation with SBB: use input index if bounds check
succeeds, otherwise use 0

sbb   r9, r9      ; set r9 to 0 if out of bounds, else
0xffffffffffffffff

and   r9d, r8d

// Array access
```

```
movsx r13d, byte [r10 + r9 + 0x10]     ; r9 would have been
sanitized to zero for out of bounds index
```

Example for Runtime/Host with AND:

For current family 6 processors (Intel® Core™, Intel Atom®, Intel® Xeon® and Intel® Xeon Phi™ processors), the Runtime/Host could use instructions that mask or range check without branching. Refer to the simple example below:

```
unsigned int user_value;

if (user_value > 255)

   return ERROR;

x = table[user_value];
```

You can make this sample code safe as shown below:

```
volatile unsigned int user_value;

If (user_value > 255)

   return ERROR;

x = table[user_value & 255];
```

This works for powers of two array lengths or bounds only. In the example above the table array length is 256 (2^8), and the valid index should be <= 255. Take care so that the compiler used doesn't optimize away the & 255 operation.

Example for Runtime/Host with CMOVcc:

A comprehensive mitigation approach to bounds check bypass (https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html) is proposed on the LLVM lists describing compiler assisted mitigation technique for LLVM using CMOVcc.

3.3 Branch target injection mitigation

Intel processors use indirect branch predictors to determine which operations are speculatively executed after a near indirect branch instruction, as shown in Table 3-1. Branch target injection (https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html), is a side channel method that takes advantage of the indirect branch predictors. By controlling the operation (“training”) of the indirect

branch predictors, attackers can cause certain instructions to be speculatively executed and then use the effects of this speculative execution for side channel analysis.

The processor uses indirect branch predictors to control only the operation of the branch instructions listed in the table below.

Table 3-1: Instructions that use Indirect Branch Predictors

Branch type	Instruction	Opcode

Near Call Indirect	`CALL r/m16,`
	`CALL r/m32,`
	`CALL r/m64`
		`FF /2`

Near Jump Indirect	`JMP r/m16,`
	`JMP r/m32,`
	`JMP r/m64`
		`FF /4`

Near Return	`RET,` `RET Imm16`	`C3,` `C2 Iw`

In this document, references to indirect branches refer only to near call indirect, near jump indirect, and near return instructions. We first cover the two general mitigation techniques available for the branch target injection method, and then discuss the runtime JIT mitigation in detail. The two general mitigation techniques enable software ecosystems to select the approach that works for their security, performance, and compatibility goals.

The Speculative Execution Side Channel Mitigations white paper (https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf) describes alternate mitigations that are not discussed here. These alternate mitigations use various methods to control the indirect branch predictor to mitigate branch target injection and bounds check bypass attacks.

Developers of managed runtimes might also consider converting near calls/jumps into far calls/jumps as a mitigation technique for branch target injection.

Section 4 lists several related security features and technologies, which are either present in existing Intel processors or are planned for future processors, and can reduce the effectiveness of the side channel methods described in this and the previous section.

3.3.1 Indirect branch control mechanisms

The first general mitigation technique for branch target injection introduces a new interface between the processor and system software. This interface provides mechanisms that allow software to mitigate exploits that attempt to influence the

victim's indirect branch predictions, such as flushing the indirect branch predictors at the appropriate time to mitigate such attacks. This mitigation strategy requires both updated system software as well as a microcode update to be loaded to support the new interface for many existing processors. We describe three new capabilities that will now be supported for this mitigation strategy in the Speculative Execution Side Channel Mitigations white paper (https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf). These capabilities are available both on existing modern Intel processors when the appropriate microcode update is applied, and on future Intel processors, which will improve the performance cost of these mitigations. In particular, the capabilities are:

- Indirect Branch Restricted Speculation (IBRS): Restricts speculation of indirect branches.
- Single Thread Indirect Branch Predictors (STIBP): Prevents indirect branch predictions from being controlled by the sibling Hyperthread.
- Indirect Branch Predictor Barrier (IBPB): Ensures that earlier code's behavior does not control later indirect branch predictions.

3.3.2 Retpoline

The second general mitigation technique for branch target injection introduces the concept of a "return trampoline", also known as *retpoline* (https://support.google.com/faqs/answer/7625886). Essentially, software replaces indirect near jump and call instructions with a code sequence that includes pushing the target of the branch in question onto the stack and then executing a return (RET) instruction to jump to that location.

Intel has worked with various open source compiler developers to ensure compiler support for retpoline, and with the OS vendors to ensure support for these mitigation techniques. For Intel® Core™ 5th generation processors (code name Broadwell) and later, this retpoline mitigation strategy also requires a microcode update for the mitigation to be fully effective.

You can find a detailed explanation of branch target injection and the mitigations available in Retpoline: A Branch Target Injection Mitigation (https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf). As stated in section 3.0 of that document:

> *Mitigations for speculation-based, side-channel security issues fall into two categories: directly manipulating speculation hardware, or indirectly controlling speculation behavior. Direct manipulation of the hardware is generally performed by microcode updates or manipulation of hardware registers. Indirect control is accomplished via software constructs that limit or constrain speculation. Retpoline is a hybrid approach since it requires updated microcode to make the speculation*

hardware behavior more predictable on some processor models. However, retpoline is primarily a software construct that leverages specific knowledge of the underlying hardware to mitigate branch target injection. The retpoline is a method to bypass the indirect branch predictor.

Retpoline is a mitigation for branch target injection on Intel processors belonging to family 6 (as enumerated by the CPUID instruction) that do not have support for enhanced IBRS. Combined with updated microcode support, retpoline (https://support.google.com/faqs/answer/7625886) can help ensure that a given indirect branch is resistant to branch target injection exploits. Retpoline sequences deliberately steer the processor's branch prediction logic to a trusted location, thereby helping to mitigate a potential exploit from steering the branch prediction logic elsewhere.

The managed runtime JIT and AOT compilers may also generate indirect branches. Therefore when the managed runtime is used in sandbox environments or where isolation is required, for microprocessors listed in the security advisory (https://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00088&languageid=en-fr), the managed runtime JIT and AOT compilers can automatically generate retpoline sequences for "near call indirect" and "near jump indirect" instead of vulnerable indirect branches.

The community is using several retpoline sequences. Generally, these sequences are functionally equivalent to each other, but differ with respect to their power impact on different microarchitectures. Inserting PAUSE and/or LFENCE instructions may reduce the power cost of the speculative spin construct in retpoline on some architectures (https://gcc.gnu.org/ml/gcc-patches/2018-01/msg01209.html).

3.3.3 Mitigation for processors with alternate empty RSB behavior

As stated in section 5.2 of Retpoline: A Branch Target Injection Mitigation (https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf):

The predictable speculative behavior of the RET instruction is the key to retpoline being a robust mitigation. RET has this behavior on all processors which are based on the Intel® microarchitecture codename Broadwell and earlier when updated with the latest microcode. Processors based on the Intel® microarchitecture codename Skylake and its close derivatives have different RSB behavior than other processors when the RSB is empty.

For processors based on Intel® microarchitecture code name Skylake and its close derivatives which have different Return Stack Buffer (RSB) behavior, although the near call indirect and near jump indirect retpolines significantly raise the bar for successful attacks, developers need to additionally use return retpolines to further mitigate the RET instruction. Refer to Example 3 in the Example retpoline sequences section below. You can identify processors with this RSB behavior using the Family/Model signatures

in Table 3-2. Refer to the CPUID Instruction (https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2a-manual.html) in the *Intel® 64 and IA-32 Architectures Developer's Manual*.

Table 3-2: Processor Family/Model signatures with alternate empty RSB behavior[3]

Family	Model
06H	4EH
06H	5EH
06H	55H
06H	66H
06H	67H
06H	8EH
06H	9EH

Managed runtimes could be used in a virtualized environment. A valuable tool in modern data centers is live migration of virtual machines (VMs) among a cluster of bare-metal hosts. However, those bare-metal hosts often differ in hardware capabilities. These differences could prevent a VM that started on one host from being migrated to another host that has different capabilities. For instance, a VM using Intel® Advanced Vector Extensions 512 (Intel® AVX512) instructions could not be live-migrated to an older system without Intel® AVX-512.

A common approach to solving this issue is exposing the oldest processor model with the smallest subset of hardware features to the VM. This addresses the live-migration issue, but results in a new issue: Software using model/family numbers from CPUID can no longer detect when it is running on a newer processor that is vulnerable to exploits of Empty RSB conditions.

To remedy this situation, a managed runtime running in a virtualized environment needs to query bit 2 of the IA32_ARCH_CAPABILITIES MSR, known as “RSB Alternate” (RSBA). Since applications can’t read MSRs directly, OS vendors may expose this information through an API to help applications take corrective measures. When RSBA is set, it indicates that the underlying VM may run on a processor vulnerable to exploits of Empty RSB conditions regardless of the processor’s Family/Model signature, and that the managed runtime should deploy appropriate mitigations.

3.3.4 Example retpoline sequences

The assembly snippets below show retpoline mitigations in GNU Assembler syntax. Refer to Retpoline: A Branch Target Injection Mitigation

[3] Additional processors may exhibit vulnerable RSB behavior that are not listed in this table.

(https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf) for more details.

Example 1: Near call/jump indirect retpoline sequence for address in register

Before:	`jmp *rax`

After:	`call load_label`
	`capture_ret_spec:`
	`pause ; lfence`
	`jmp capture_ret_spec`
	`load_label:`
	`mov %rax, (%rsp)`
	`ret`

Before:	`call *rax`

After:	`jmp label2`
	`label0:`
	`call label1`
	`capture_ret_spec:`
	`pause ; lfence`
	`jmp capture_ret_spec`
	`label1:`
	`mov %rax, (%rsp)`
	`ret`
	`label2:`
	`call label0`

Example 2: Near call/jump indirect retpoline sequence for address in memory

Before:	`jmp *mem`

After:	`push mem`
	`call load_label`
	`capture_ret_spec:`
	`pause ; lfence`
	`jmp capture_ret_spec`
	`load_label:`
	`lea 8(%rsp), %rsp`

	ret

Before:	call *mem
After:	jmp label2 label0: push mem call label1 capture_ret_spec: pause ; lfence jmp capture_ret_spec label1: lea 8(%rsp), %rsp ret label2: call label0

Example 3: Near return retpoline sequence for processors with alternate empty RSB behavior

Before:	ret
After:	call load_label capture_ret_spec: pause ; lfence jmp capture_ret_spec load_label: lea 8(%rsp), %rsp ret

3.3.5 Runtime/Host retpoline mitigation overview

Managed runtimes and host application are usually implemented in C/C++. To mitigate branch target injection when the managed runtime is used in a sandbox environment, you should also consider applying mitigations to the runtime itself, the hosting process, and the libraries used by the runtime.

Managed runtimes can also have inline assembly. Some low-level critical elements of the managed runtimes are implemented in assembly for performance reasons (for example, interpreter, portions of object allocation or garbage collection, etc.). The

mitigations for branch target injection need to be carefully applied to the inline assembly as well.

In the following sections, we outline the compiler flags that you can use to mitigate branch target injection when you compile managed runtime C/C++ elements to create a sandbox environment.

3.3.6 Linux* and GCC retpoline mitigation

Managed runtimes written in C/C++ can use mitigation support in native C/C++ compilers. For example, you can implement branch target injection mitigation on Linux* with GCC (https://gcc.gnu.org/git/?p=gcc.git;a=commit;h=c6b72be421ded17e0c156070ba6e90aa6c335ed6) by compiling the runtime with the following compiler switches:

```
-mindirect-branch=thunk/thunk-inline/thunk-extern
```

When compiling for the processors identified in Table 3-2 above, you also need to generate the `ret` retpolines (https://gcc.gnu.org/git/?p=gcc.git;a=commit;h=357311dd400f7f72d2132f2f94161ece39bf08c6) with the following compiler switches:

```
-mfunction-return=thunk/thunk-inline/thunk-extern
```

3.3.7 Intel® C Compiler (ICC) retpoline mitigation

Intel® C Compiler v18.0.3 also supports the switches described above. Work is ongoing to backport these switches version 16 and version 17.

3.3.8 LLVM Clang retpoline mitigation

The LLVM Clang C/C++ Compiler supports `-mretpoline` and `-mretpoline-external-thunk` (https://reviews.llvm.org/rC323155) switches for retpoline generation. These switches are backported to version 6.0.0 (http://releases.llvm.org/6.0.0/docs/ReleaseNotes.html), and work is ongoing to backport to version 5.0.2 (http://lists.llvm.org/pipermail/llvm-dev/2018-March/121771.html).

3.3.9 Speculation execution using return stack buffers

The RSB is a last-in-first-out (LIFO), fixed-size stack that is implemented in hardware. In the RSB, CALL instructions “push” entries, and RET instructions “pop” entries. Prediction of RET instructions relies on the RSB first and differs from prediction of JMP and CALL instructions. On many Intel processors, enough return instructions in a row can cause the RSB to wrap and to incorrectly speculate to a previous RET instruction’s target.

For example, if the RSB pointer is pointing to the bottom entry of the RSB and two return instructions are executed, the RSB pointer will wrap to the top entry. Therefore, the first return address prediction will be taken from the bottom entry, and the second return address prediction will be taken from the top entry.

Intel® processors based on CPUID family 6, excluding the processors listed above in table 3-2, exhibit the RSB wrap behavior. For these processors, the following techniques may be applied for mitigation.

One mitigation for speculative execution exploits targeting the RSB is tracking the call depth and using RSB stuffing whenever the call depth exceeds 16 entries, which could cause the RSB pointer to wrap. An example RSB stuffing sequence is shown below:

```
void rsb_stuff(void) {
            asm(".rept 16\n"
                  "call 1f\n"
                  "pause ; lfence\n"
                  "1: \n"
                  ".endr\n"
                  "addq $(8 * 16), %rsp\n");
}
```

For more details on RSB stuffing, refer to section 5.0 of the Retpoline: A Branch Target Injection Mitigation (https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf) white paper.

On Intel processors belonging to family 6 (as enumerated by the CPUID instruction) that do not have support for enhanced IBRS, you can use *near return* retpoline to mitigate speculative execution exploits that target the RSB. Combined with updated microcode support, near return retpoline can help ensure that a given return instruction is resistant to speculative execution exploits using the RSB. Near return etpoline sequences deliberately steer the processor's prediction logic to a known location, thereby helping to mitigate potential exploits from steering the prediction logic elsewhere. For more details, refer to the near return retpoline sequence in Example 3 of section 3.3.4.

3.4 Speculative store bypass mitigation

Modern high performance processors use memory disambiguation predictors to speculatively execute load operations even when the addresses of preceding store operations are unknown. In cases where the memory disambiguation predictor does

not correctly predict an address overlap, speculative loads could consume stale data until the processor makes the necessary corrections.

Refer to the example scenario below:

```
//Store N at address ‘ptrA’

*ptrA = N;

 ...

//expr is a high-latency expression evaluating to address
‘prtA’

*(<expr>) = P;

//Read value from ‘ptrA’

X = *ptrA; //X can read the stale value N

//Leak the value X into the cache using a gadget.
```

For additional information, refer to the Intel Analysis of Speculative Execution Side Channels white paper (https://software.intel.com/sites/default/files/managed/b9/f9/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf).

3.4.1 Speculative store bypass mitigation

Managed runtimes using language-based security in sandbox environments are vulnerable to malicious actors who can use confused deputy code to influence JIT/AOT code generation to break the sandbox isolation. Software that does not rely on language-based security mechanisms, for example, because it instead uses process isolation, might not need the speculative store bypass mitigation. The speculative store bypass method can be mitigated with software modifications or using the processor-supported mitigation mechanism.

3.4.2 Software-based mitigations

Isolating secrets to a separate address space from untrusted code will mitigate speculative store bypass. For example, creating separate processes for different websites ensures that secrets are mapped to different address spaces than a malicious website executing code. Similar techniques can be used for other runtime environments that rely on language-based security to run trusted and untrusted code within the same process.

Inserting LFENCE between a store and a subsequent load, or between the load and any subsequent usage of the data returned that might create a side channel will mitigate

speculative store bypass. Software should apply this mitigation when there is a realistic risk of an exploit to minimize performance degradation.

3.4.3 Speculative Store Bypass Disable (SSBD) overview

When the software based mitigations are not feasible, you can use the processor-supported Speculative Store Bypass Disable (SSBD) mechanism to mitigate speculative store bypass. When SSBD is set, loads will not execute speculatively until the addresses of the older stores are known. Use SSBD judiciously to minimize the impact on performance. Managed runtimes can use OS-provided API's to enable and disable SSBD.

For additional information on SSBD, refer to the Speculative Execution Side Channel Mitigations document (https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf).

3.4.4 Speculative Store Bypass Disable (SSBD) on Linux

In Linux, the `spec_store_bypass_disable` boot option controls whether the system uses the SSBD optimization. The parameter takes the following values:

`on`	Unconditionally disable speculative store bypass
`off`	Unconditionally enable speculative store bypass
`auto`	The kernel detects whether the CPU model contains an implementation of speculative store bypass and picks the most appropriate mitigation. If the CPU is not vulnerable, this option selects off. If the CPU is vulnerable, the default mitigation is architecture and Kconfig dependent. See below.
`prctl`	Controls speculative store bypass per thread via `prctl`. Speculative store bypass is enabled for processes by default. The state of the control is inherited on fork.
`seccomp`	Same as `prctl`, except all `seccomp` threads will disable Speculative store bypass unless they explicitly opt out

Not specifying this boot option is equivalent to setting `spec_store_bypass_disable=auto`.

Setting the `spec_store_bypass_disable` parameter value to `prctl` or `seccomp` enables controlling speculative store bypass per thread. The `prctl` interface is documented in the Linux kernel documentation

(https://github.com/torvalds/linux/blob/master/Documentation/userspace-api/spec_ctrl.rst).

`prctl` has two options that are related to speculative store bypass disable:

- `PR_GET_SPECULATION_CTRL`
- `PR_SET_SPECULATION_CTRL`

`PR_GET_SPECULATION_CTRL` returns the state of the speculation feature selected with argument 2 of `prctl`(2). The return value uses bits 0-3, corresponding to the following:

Bit	Definition	Description
0	PR_SPEC_PRCTL	Mitigation can be controlled per task using `PR_SET_SPECULATION_CTRL`.
1	PR_SPEC_ENABLE	The speculation feature is enabled, mitigation is disabled.
2	PR_SPEC_DISABLE	The speculation feature is disabled, mitigation is enabled.
3	PR_SPEC_FORCE_DISABLE	Same as PR_SPEC_DISABLE, but cannot be undone. A subsequent `prctl(...,` `PR_SPEC_ENABLE)` will fail.

`PR_SET_SPECULATION_CTRL` allows controlling the speculation feature.

Sample invocations:

- `prctl(PR_GET_SPECULATION_CTRL, PR_SPEC_STORE_BYPASS, 0, 0, 0);`
- `prctl(PR_SET_SPECULATION_CTRL, PR_SPEC_STORE_BYPASS, PR_SPEC_ENABLE, 0, 0);`
- `prctl(PR_SET_SPECULATION_CTRL, PR_SPEC_STORE_BYPASS, PR_SPEC_DISABLE, 0, 0);`
- `prctl(PR_SET_SPECULATION_CTRL, PR_SPEC_STORE_BYPASS, PR_SPEC_FORCE_DISABLE, 0, 0);`

3.4.5 Speculative Store Bypass Disable (SSBD) on Windows*

Microsoft plans to provide a mitigation that leverages the new hardware features in a future Windows* update. More details are available on Microsoft's public blog (https://blogs.technet.microsoft.com/srd/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass-cve-2018-3639/).

4.0 Related Intel security features and technologies

Several related security features and technologies, which are either present in existing Intel processors or are planned for future processors, can reduce the effectiveness of the side channel methods mentioned in the previous sections.

4.1 Execute Disable Bit

Execute Disable Bit is a hardware-based security feature present in existing Intel processors that can help reduce system exposure to viruses and malicious code. Execute Disable Bit allows the processor to classify areas in memory where application code can or cannot execute, even speculatively. This helps reduce the gadget space, which increases the difficulty of branch target injection attacks. All major operating systems enable Execute Disable Bit support by default.

While generating code, managed runtime JIT/AOT compilers can mark the code buffers as not executable until the compilation of the method is complete. This is good code hygiene in general, but is especially helpful in mitigating gadget sources in managed runtimes.

4.2 Control flow Enforcement Technology (CET)

On future Intel processors, Control flow Enforcement Technology (CET) will allow developers to limit near indirect jump and call instructions to only target ENDBRANCH instructions. This feature can help reduce the speculation allowed to instructions that are not ENDBRANCH instructions. This greatly reduces the gadget space, which increases the difficulty of branch target injection attacks.

CET also provides capabilities to defend against Return-Oriented-Programming (ROP) control-flow subversion attacks. However, the retpoline technique closely resembles the approaches used in ROP attacks. Be aware that if used in conjunction with CET, retpoline might trigger false positives in the CET defenses.

For additional information on CET, refer to the Control-flow Enforcement Technology Preview (https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technologypreview.pdf).

4.3 Protection keys

On future Intel processors that have both hardware support for mitigating rogue data cache load and protection keys support, protection keys can limit the data that is

accessible to a piece of software. This can be used to limit the memory addresses that could be revealed by a bound check bypass or branch target injection attack.

A managed runtime could deploy protection keys by changing the protection key used to map secrets and then limiting what is executed while those secrets are marked accessible in the Protection Keys Rights register (PKRU). The secrets being protected could be of any class: runtime host, runtime environment or payload.

Protection keys can also be used to produce execute-only memory areas. An execute-only memory area cannot be accessed with loads and stores, which potentially limits gadget discovery which might then be used to carry out an exploit.

As is the case any time protection keys are in use, gadgets containing the WRPKRU instruction are valuable in defeating mitigations provided by protection keys. Managed runtimes should limit available occurrences of WRPKRU in both their own executables and the generated instructions.

4.4 Trusted execution environments (TEE)

The goal of all side channel method attacks is to steal secrets in the OS and application memory or register states. Removing secret data from the normal OS/application security domains would remove the motivation for those attacks. Intel provides Intel® Software Guard Extensions (Intel® SGX) and Intel® Trusted Execution Engine (Intel® TXE) on some Intel processors to provide a trusted execution environment for parts of applications that need to protect their data or algorithms. We encourage application developers to partition their applications into normal and secure portions, and then move the portions of their application that contain secrets and must be secure into a trusted execution environment.

5.0 References

- Google* Project Zero description of bounds check bypass (Spectre variant 1) and branch target injection (Spectre variant 2) (https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html)
- Retpoline: A Branch Target Injection Mitigation (https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf)
- Google retpoline introduction (https://support.google.com/faqs/answer/7625886)
- Intel security advisory regarding models and steppings for retpoline (https://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00088&languageid=en-fr)
- CPUID reference (https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2a-manual.html)
- Side-Channel Security Issue: Intel® Software Support (https://software.intel.com/en-us/side-channel-security-support)
- Speculative Execution Side Channel Mitigations (https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf)
- Intel Analysis of Speculative Execution Side Channels (https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf)
- Site Isolation (https://www.chromium.org/Home/chromium-security/site-isolation)
- Mozilla Foundation Security Advisory 2018-01 (https://www.chromium.org/Home/chromium-security/site-isolation)
- Linux Speculation Control (https://github.com/torvalds/linux/blob/master/Documentation/userspace-api/spec_ctrl.rst)
- Analysis and mitigation of speculative store bypass (CVE-2018-3639) (https://blogs.technet.microsoft.com/srd/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass-cve-2018-3639/)
- GCC support
  - https://gcc.gnu.org/ml/gcc-patches/2018-01/msg01209.html
  - https://gcc.gnu.org/git/?p=gcc.git;a=commit;h=c6b72be421ded17e0c156070ba6e90aa6c335ed6
  - https://gcc.gnu.org/git/?p=gcc.git;a=commit;h=357311dd400f7f72d2132f2f94161ece39bf08c6
- LLVM compiler support
  - https://reviews.llvm.org/rC323155
  - http://releases.llvm.org/6.0.0/docs/ReleaseNotes.html
  - http://lists.llvm.org/pipermail/llvm-dev/2018-March/121771.html

    - https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html
- Microsoft compiler options and guidance
    - https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/
    - https://docs.microsoft.com/en-us/cpp/build/reference/qspectre
    - https://docs.microsoft.com/en-us/cpp/cpp/spectre
- Spectre Mitigations in Microsoft C/C++ Compiler (https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html)
- Control-Flow Enforcement Technology Preview (https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf)